Politiques d'adaptation pour la reconfiguration du composant de localisation

Julien Dormoy — Olga Kouchnarenko — Hassan Mountassir

Laboratoire d'Informatique de Franche-Comté - Université de Franche-Comté
16, route de Gray F-25030 Besançon Cedex
{jdormoy,okouchnarenko,hmountassir}@lifc.univ-fcomte.fr

RÉSUMÉ. *Les approches à base de composants sont intensivement étudiées dans le cadre des systèmes complexes. Ces approches visent à concevoir des systèmes et des applications par assemblage de composants préfabriqués, réutilisables et faciles à maintenir. Afin de répondre à des besoins spécifiques, une des approches est l'utilisation de politiques d'adaptation permettant de reconfigurer dynamiquement le modèle à composants par rapport au contexte de son environnement. Le travail présenté dans cet article repose sur un cadre formel permettant de décrire des politiques d'adaptation appliquées à un composant de localisation. Ce composant, que nous spécifions en Fractal, permet de fournir une position optimale, obtenue à partir de plusieurs positions fournies par plusieurs systèmes de localisation (GPS, Wifi). Nous définissons deux politiques d'adaptation pour ce composant et simulons son fonctionnement sur une extension de Fractal. Suite aux expérimentations, nous proposons une extension des politiques d'adaptation pour prendre en compte de nouveaux aspects non fonctionnels.*

ABSTRACT. *Component-based development becomes commonly used technique to build complex systems. In order to satisfy specific requirements, adaptation policies are a solution to dynamically reconfigure the component model in relation to the context of its environment. The work presented in this paper is based on a formal way to describe adaptation policies applied to a component of localization. This component, which we specify in Fractal, provides an optimal position. We define two adaptation policies for this component and simulate its functioning on an extension of Fractal. After experimentations, we propose to extend adaptation policies to consider new non-functional aspects.*

MOTS-CLÉS : *Composants, Fractal, Politique d'adaptation, Reconfiguration dynamique*

KEYWORDS: *Components, Fractal, Adaptation Policy, Dynamic Reconfiguration*

JESA – 43/2009. MSR 2009, pages 773 à 787

1. Introduction

Les systèmes à base de composants sont intensivement étudiés ces dernières années, ils sont conçus en combinant des briques matérielles et/ou logicielles réutilisables, appelées composants. L'approche par composants vise à diminuer la complexité de conception et de développement de systèmes. Son succès dépend de l'assemblage des composants connectés, i.e., de leur capacité à communiquer et à coopérer malgré les différences dans les langages d'implantation, dans leurs environnements d'exécution, etc.

Le travail présenté dans cet article s'est déroulé dans le cadre du projet ANR TACOS[1]. L'objectif de ce projet est de proposer une approche par composants pour la spécification de systèmes sûrs, depuis l'expression des besoins jusqu'à une spécification formelle, en utilisant ou adaptant des langages et des outils existants. Le domaine d'application est celui du transport terrestre. Les systèmes considérés, à la fois distribués et embarqués, nécessitent l'expression de propriétés fonctionnelles et non-fonctionnelles, incluant entre autres des contraintes temporelles et de disponibilité.

Parmi les verrous scientifiques identifiés dans les approches par composants existantes, nous nous intéressons à l'identification d'un bon niveau d'abstraction pour spécifier l'architecture et le comportement d'un assemblage de composants pour des contextes préalablement identifiés. Un composant pouvant lui-même être défini comme une composition de composants existants, sa spécification doit contenir les informations nécessaires à la coopération entre les divers composants. Une question se pose alors : comment déterminer statiquement ou dynamiquement si l'intégration/remplacement d'un composant dans l'architecture d'un système peut être acceptée ou refusée ? Les langages de haut niveau pour spécifier les composants doivent être suffisamment riches pour exprimer des mécanismes d'assemblage entre composants et permettre une vérification de la compatibilité ou de l'interopérabilité de composants.

L'objectif de ce papier est d'apporter des éléments de réponse à ces questions via une étude de cas : le composant de localisation. Ce composant est un des composants du CyCab, un véhicule électrique autonome pouvant se déplacer seul ou dans un convoi de véhicules. Le point de vue adopté sur les composants est celui de (Szyperski, 1999), où un composant est une unité de composition logicielle décrite par ses interfaces. Pour cette étude de cas, nous avons choisi d'utiliser les travaux de (Chauvel *et al.*, 2009) portant sur la composition de politiques d'adaptation pour les composants Fractal (Bruneton *et al.*, 2004; Bruneton *et al.*, 2006). De plus, en vue d'exprimer d'autres aspects non-fonctionnels, nous exploitons les travaux sur les logiques reposant sur les évènements (Lee *et al.*, 1999; Gonnord *et al.*, 2009).

Contributions. Dans cet article, nous proposons et implantons une architecture Fractal d'une étude de cas portant sur le composant de localisation. Nous définissons des politiques d'adaptation permettant des reconfigurations architecturales tenant compte d'aspects non-fonctionnels. Nous mettons en place ces politiques pour simu-

1. Le projet TACOS (ANR-06-SETI-017) est financé par l'ANR (http ://tacos.loria.fr).

ler les mécanismes d'assemblage et pour observer le fonctionnement du composant de localisation. Nous proposons une extension des politiques d'adaptation pour tenir compte des évènements des différents composants.

Organisation du papier. Les travaux connexes sont présentés dans la section 2. La section 3 introduit le composant de localisation ainsi que son architecture. Dans la section 4, nous présentons les politiques d'adaptation pour ensuite les appliquer et expérimenter sur le cas d'étude. Nous proposons dans la section 5 d'enrichir les politiques d'adaptation avec la prise en compte d'évènements avant de conclure dans la section 6.

2. Travaux connexes

Les travaux présentés dans (Chauvel, 2008) identifient le cycle d'adaptation comme composé d'une partie d'observation, d'une partie de décision, et d'une partie d'intro-action. Pour combiner ces trois parties, des politiques d'adaptation de haut niveau (Chauvel *et al.*, 2009) ont été définies. Elles utilisent la logique floue (Zadeh, 1975) comme moteur de décision pour décrire ces politiques de manière qualitative. Une première version d'un simulateur a été développé en Kermeta[2] mais a présenté des problèmes de performance dû à l'intégration de la logique floue et des politiques d'adaptation. Le choix des auteurs s'est alors porté sur Fractal qui intègre un langage d'intro-action ainsi qu'une extension (COSMOS (Conan *et al.*, 2007) ou WildCat (David *et al.*, 2005)) dédiée à l'observation du contexte mais ne dispose pas d'un moteur de décision capable de faire le lien entre les deux. Fractal offre la possibilité de développer une version « pure Java » qui permet de pallier aux problèmes de performance. Un moteur de logique floue a alors été développé dans une extension de Fractal nommée Tangram4Fractal.

D'autres travaux dans (Lee *et al.*, 1999) présentent une plate-forme appelée MaC (pour Monitoring and Checking) développée dans le but de s'assurer, lors de l'exécution, que le comportement du système se déroule selon un ensemble de besoins spécifiés formellement. Cette plate-forme s'appuie sur une logique à base d'évènements appelée EDL (pour *Event Definition Language*).

Dans (Tournier *et al.*, 2005; Gonnord *et al.*, 2009), les auteurs proposent une architecture logicielle à composants pour la gestion de la qualité de service au niveau des ressources, appelée Qinna. Ce travail intègre plus particulièrement une extension de la logique EDL, qMEDL pour exprimer, en plus des évènements, des aspects quantitatifs.

Les systèmes embarqués peuvent être considérés commes des composants autonomes et communicants qui intègrent à la fois une partie logicielle et une partie matérielle. Ils se caractérisent par une interaction continue avec leur environnement. De nombreuses applications dédiées aux systèmes embarqués sont utilisées

2. http ://www.kermeta.org/

dans le domaine du transport, des télécommunications, ainsi que dans de nombreux produits électroniques. Ces systèmes, dont les ressources sont généralement limitées, doivent satisfaire non seulement des exigences fonctionnelles mais également des exigences non fonctionnelles pour optimiser l'utilisation de ces ressources (énergie, mémoire, ...). Dans le contexte du modèle Fractal, certains travaux (Navas *et al.*, 2009; Polakovic *et al.*, 2007) proposent des stratégies de reconfiguration dynamique pour optimiser la taille mémoire utilisée grâce à une implémentation en C de Fractal pour l'embarqué, nommée Think[3]. Think sépare les problèmes liés à l'architecture et les problèmes liés au développement ce qui facilite la portabilité, la réutilisation et l'optimisation du code lors du déploiement. Cette approche apporte aussi une grande flexibilité lors de l'exécution puisque les performances du système peuvent être optimisées en adaptant uniquement l'architecture sans modifier le code des composants. De plus, Think dispose d'une librairie de composants nommée Kortex fournissant des services fréquemment utilisés dans les systèmes embarqués.

3. Motivations

3.1. *Le composant de localisation*

Depuis plusieurs années, des programmes de recherche se sont intéressés au concept de véhicules intelligents. Ces véhicules sont dotés de capteurs qui leur permettent de percevoir leur environnement pour pouvoir assister le conducteur, gérer des situations d'urgences et se déplacer en évitant des obstacles. L'objectif de ces projets est de produire des véhicules sûrs et complètement autonomes pour permettre des déplacements individuels ou collectifs d'un point à un autre.

Dans ce cadre, l'INRIA Rhône-Alpes a développé une première version d'un système de transport urbain nommé CyCab de véhicules en libre-service. Ce système de transport public est basé sur un convoi de véhicules électriques et conçu tout particulièrement pour des zones où la circulation automobile doit être limitée : hypercentre urbain, gare/aérogare, campus universitaire, site touristique. Ces véhicules sont conçus pour une conduite simplifiée et sécurisée. Un terminal multimédia embarqué et connecté à Internet permet d'obtenir des renseignements touristiques ou commerciaux sur la ville.

Plusieurs versions du CyCab existent selon une conduite automatique ou semi-automatique. Dans cet article, nous nous intéressons au CyCab avec conduite automatique et plus particulièrement à un de ses sous-composants : le composant de localisation. Ce composant permet de localiser le véhicule à tout moment. La conduite étant automatique, le déplacement du véhicule ne peut se faire sans ce composant. Pour avoir une meilleure confiance dans la position du véhicule, le composant de localisation utilise plusieurs systèmes de localisation (GPS, Wifi...) et fusionne les positions récupérées. De plus, il peut faire appel à d'autres sous-composants du CyCab (Ac-

3. http ://think.ow2.org/

céléromètre, Capteur de vitesse…) qui permettent d'avoir une position relative à la position précédente. Cela permet de valider les positions données par les systèmes de localisation et détecter celles qui ne sont pas cohérentes.

Le fait d'avoir plusieurs systèmes de localisation et de valider les positions, demande l'utilisation de beaucoup d'énergie. Pour trouver un compromis entre la consommation d'énergie et l'obtention d'une position la plus correcte possible, nous souhaiterions adapter le composant de localisation en fonction de l'énergie disponible et de la confiance en nos systèmes de localisation (par exemple, le nombre de satellites disponibles pour le GPS). Cette adaptation peut être faite en reconfigurant (ajouter ou supprimer dynamiquement) les systèmes soit de localisation, soit de validation des positions.

3.2. *Architecture du composant de localisation*

Nos travaux se situent dans le cadre de la plate-forme Fractal (Bruneton *et al.*, 2004; Bruneton *et al.*, 2006) qui est un modèle hiérarchique permettant la définition, le contrôle et la reconfiguration dynamique d'une architecture à base de composants. Cette plate-forme est extensible et permet d'observer, de contrôler, ou de modifier dynamiquement une architecture.

La figure 1 présente notre proposition d'architecture Fractal pour le composant de localisation. Nous avons fait le choix de ne modéliser que deux systèmes de localisation (GPS et Wifi). Le composant *Ctrl* reçoit les demandes de localisation du CyCab. Il démarre alors en parallèle les composants *GPS* et *Wifi* qui vont localiser le véhicule. Une fois les deux positions récupérées grâce à `locate1` et `locate2` par le composant *Ctrl*, il les valide grâce au composant *Check*. Une fois la validation effectuée, le composant *Ctrl* fusionne les positions et renvoie la position résultante.

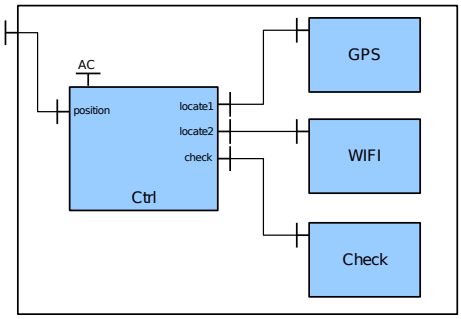

Figure 1. *Architecture du composant de localisation en Fractal*

Nous souhaitons pouvoir adapter dynamiquement l'architecture du composant de localisation en fonction de l'énergie restante et de la confiance accordée aux systèmes de localisation. Cette adaptation doit obéir aux deux règles suivantes :

1) Le composant *Wifi* n'est déployé que si l'énergie disponible est élevée.

2) Le composant *Check* ne doit valider les positions que lorsque la confiance dans les systèmes de localisation est faible.

4. Politiques d'adaptation appliquée au composant de localisation

4.1. *Les politiques d'adaptation*

Les travaux présentés dans (Chauvel *et al.*, 2009) proposent une extension de la plate-forme Fractal et de son implantation de référence Julia[4] avec les mécanismes nécessaires à l'exécution de politiques d'adaptation de haut niveau. Ces politiques sont interprétées grâce à un moteur prenant en charge l'interprétation de propriétés exprimées en logique floue (Zadeh, 1975). Cette logique utilise la notion d'ensembles flous dont l'appartenance est graduelle par opposition aux ensembles classiques dont l'appartenance est stricte. En logique floue, un élément peut appartenir plus ou moins à un ensemble. La logique floue offre également un mécanisme appelé contrôle flou (Pedrycz, 1993) qui permet d'inférer des valeurs réelles à partir de descriptions qualitatives. La politique d'adaptation est alors représentée sous la forme d'un ensemble de règles qualitatives qui peuvent soit impacter l'architecture du modèle à composants, soit impacter localement un composant en modifiant la valeur d'un de ses paramètres. Plusieurs éléments définissent une politique d'adaptation :

– Les reconfigurations architecturales qui permettent de spécifier les différentes modifications de l'architecture. Ces reconfigurations architecturales peuvent être faites grâce à des actions architecturales. Elles sont écrites avec FScript[5] qui permet de reconfigurer dynamiquement une architecture. Notons que la terminaison d'une reconfiguration est garantie grâce aux restrictions du langage FScript. Nous n'avons pas identifié de moment critique et les reconfigurations peuvent se produire à tout moment.

– Les propriétés et leurs domaines qui permettent de capturer l'environnement du système. Le domaine définit le vocabulaire spécifique à utiliser pour qualifier les propriétés qui lui sont associées.

– Les règles de reconfigurations architecturales qui permettent de mettre en relation le contexte d'exécution du système et l'utilité de déclencher une action architecturale.

– Les règles de configurations locales qui permettent de spécifier l'évolution souhaitée des propriétés locales d'un composant.

L'extension de Fractal proposée dans (Chauvel *et al.*, 2009) prend en compte ces politiques d'adaptation et les composent pour décider des différentes actions à mener pendant l'exécution du système. Les règles de reconfiguration architecturales sont

4. http ://fractal.objectweb.org/julia/index.html
5. http ://fractal.ow2.org/fscript/

évaluées en fonction du degré d'appartenance des propriétés à leur domaine. Chaque règle est alors associée à une utilité de reconfiguration. La reconfiguration architecturale correspondant à la plus grande utilité est alors exécutée. Dans le cas où deux utilités seraient égales, une des règles est arbitrairement exécutée.

4.2. *Application au composant de localisation*

Notre objectif est de définir et de mettre en oeuvre les politiques d'adaptation pour le composant de localisation. Nous devons définir deux politiques qui vont porter l'une sur l'ajout ou la suppression du composant *Wifi* et l'autre sur l'ajout ou la suppression du composant *Check*. Une fois ces deux politiques d'adaptation définies, nous exploitons l'extension de la plate-forme Fractal pour les composer et modifier automatiquement l'architecture lorsque celle-ci est exécutée. Dans la suite de cette section, nous présentons les différentes étapes permettant de créer la politique d'adaptation liée au composant *Wifi*. Parmi les éléments définissant une politique d'adaptation ci-dessus, nous utilisons les trois premiers éléments. Les règles de configurations locales ne nous sont pas utiles car nous ne modifions pas les propriétés locales de nos composants. Pour créer la politique d'adaptation, il faut rajouter tous les éléments qui vont suivre dans un fichier.

4.2.1. *Reconfigurations architecturales*

Nous devons tout d'abord définir les différentes reconfigurations qui pourront être utilisées par la politique d'adaptation. Nous définissons, grâce à FScript, les actions architecturales permettant de modifier l'architecture du modèle à composants. Dans notre cas, deux actions sont possibles :

1) L'ajout du composant *Wifi* ;

2) La suppression du composant *Wifi*.

L'ajout d'un composant consiste à créer une instance du nouveau composant (`new`) et de le nommer (`set-name`). Ensuite, le composant est intégré à l'architecture (`add`) puis il est lié avec les autres composants (`bind`). Enfin, le nouveau composant est démarré (`start`). Le code FScript correspondant à l'ajout du composant *Wifi* est présenté ci-dessous :

```
action addWifi(root){
  newWifi = new("location.Wifi");
  set-name($newWifi, "wifi");
  add($root, $newWifi);
  bind($root/child::Ctrl/interface::locate2,
       $newWifi/interface::loc);
  start($newWifi);
}
```

La suppression d'un composant consiste à arrêter les composants liés (`stop`) puis à supprimer les liens avec ces composants (`unbind`). Ensuite, le composant est

supprimé (`remove`) et les composants qui ont été arrêtés sont redémarrés (`start`). Le code FScript correspondant à la suppression du composant *Wifi* est présenté ci-dessous :

```
action removeWifi(root){
  stop($root/child::ctrl);
  unbind($root/child::ctrl/interface::locate2);
  remove($root, $root/child::wifi);
  start($root/child::ctrl);
}
```

Notons que le langage FScript peut aussi être utilisé pour activer ou désactiver un composant plutôt que de l'ajouter ou de le supprimer complètement du modèle. Le fait d'ajouter ou de supprimer un composant libère de la mémoire vive et peut être intéressant si le système embarqué dispose de peu de mémoire. Cela pourrait permettre, par exemple, d'augmenter la vitesse de calcul de la position du véhicule grâce au Wifi et donc de le localiser plus souvent lorsque le GPS n'est pas disponible. Par contre, si la mémoire vive est assez grande, une activation ou désactivation du composant peut suffire.

Pour faire le lien entre la politique d'adaptation et les actions architecturales, nous les intégrons dans le fichier de politique de la manière suivante :

```
policy withWifi
is
  reconfiguration addWifi    is "addWifi.fscript"
  reconfiguration removeWifi is "removeWifi.fscript"
```

4.2.2. *Les propriétés et leurs domaines*

Nous devons ensuite définir un ensemble de propriétés permettant de capturer l'environnement du système ainsi que leurs domaines. Pour notre cas d'étude, nous avons besoin d'une propriété permettant de capturer le niveau d'énergie du CyCab. Pour que la politique d'adaptation puisse accèder à cette donnée, nous ajoutons un attribut sur le composant *Ctrl* permettant de récupérer le niveau d'énergie actuel. Le domaine de cette propriété évolue entre 0 % et 100 % et sera qualifié grâce aux termes `low`, `medium` et `high`. L'utilisation de `sensor` ou de `actuator` permet à la politique d'adaptation de récupérer ou de modifier le niveau d'énergie à tout moment (`actuator` sera utilisé lors de la reconfiguration locale). La propriété est alors définie de la manière suivante dans le fichier de politique :

```
property power : Real
evolves in [0, 100] as 'low' 'medium' 'high'
sensor is getPower on Ctrl
```

4.2.3. *Règles de reconfigurations architecturales*

Nous définissons maintenant les règles de reconfigurations architecturales qui permettent de spécifier l'utilité de reconfigurer l'architecture en fonction du contexte

d'exécution. Le code ci-dessous donne les règles de reconfigurations architecturales (voir section 3.2) à ajouter dans le fichier de politique.

```
  #Adding Wifi
  when power is 'high' or 'medium'
    if size($context/child::wifi) == 0
    then utility of addWifi is 'high'

  when power is 'low'
    if size($context/child::wifi) == 0
    then utility of addWifi is 'low'

  # Removing Wifi
  when power is 'high'
    if size($context/child::wifi) > 0
    then utility of removeWifi is 'low'

  when power is 'medium' or 'low'
    if size($context/child::wifi) > 0
    then utility of removeWifi is 'high'
end policy
```

Dans ces règles, l'utilisation de `when` permet de capturer le contexte d'exécution. La partie comportant `if` permet de tester si le composant *Wifi* est présent dans l'architecture courante. La partie *utility* permet de spécifier le niveau (`low`, `medium` ou `high`) du besoin d'adaptation de l'architecture par rapport au contexte d'exécution en cours.

4.3. *Expérimentations*

Dans cette partie, nous présentons la simulation de l'exécution du composant de localisation. Lors de l'exécution, deux politiques d'adapation sont prises en compte et mises en oeuvre. Le première concerne la gestion de composant *Wifi* en fonction du niveau d'énergie comme présenté dans la section 4.2. La deuxième politique concerne la gestion du composant *Check* en fonction de la confiance donnée aux systèmes de localisation. La confiance est calculée grâce au nombre de satellites utilisés par le GPS mais aussi avec la présence ou non du composant *Wifi*. Nous mettons en place la politique correspondante ci-dessous :

```
policy WithCheck
is
  reconfiguration addCheck is "addCheck.fscript"
  reconfiguration removeCheck is "removeCheck.fscript"

  property trust : Real
    evolves in [0, 100] as 'low' 'medium' 'high'
    sensor is getTrust on Ctrl
```

```
  when trust is 'high'
    if size($context/child::check) == 0
    then utility of addCheck is 'low'

  when trust is 'medium' or 'low'
    if size($context/child::check) == 0
    then utility of addCheck is 'high'

  when trust is 'high'
    if size($context/child::check) > 0
    then utility of removeCheck is 'high'

  when trust is 'medium' or 'low'
    if size($context/child::check) > 0
    then utility of removeCheck is 'low'

end policy
```

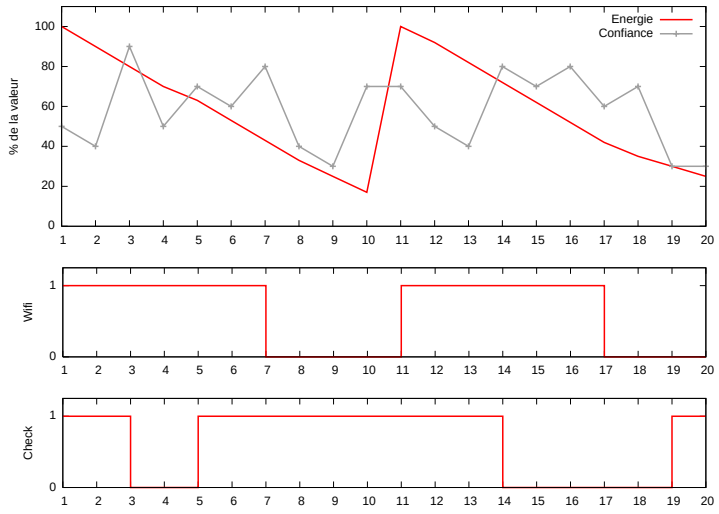

Figure 2. *Expérimentation avec le composant de localisation*

La figure 2 présente les résultats de la simulation. Le premier graphique présente l'évolution de la quantité d'énergie disponible et l'évolution de la confiance dans les systèmes de localisation au fil des étapes de la simulation. Entre les étapes 10 et 11, nous avons simulé la recharge des batteries du véhicule. Les deux autres graphiques permettent de montrer si les composants *Wifi* et *Check* sont présents dans l'architecture (0 : absent, 1 : présent). Cette simulation met en évidence que lorsque le niveau d'énergie baisse, le composant *Wifi* est supprimé de l'architecture. On peut noter que la composition des politiques d'adaptation implantée dans l'extension de Fractal permet

de faire un choix sur la reconfiguration à effectuer. Par exemple, dans l'étape 7, le niveau d'énergie étant faible et la confiance étant élevée, les deux politiques peuvent être appliquées. L'application choisit alors l'adaptation la plus appropriée et la déclenche. Dans ce cas, ce sera le composant *Wifi* qui sera supprimé.

5. Prise en compte des événements

L'étude de cas ci-dessus nous a permis de reconfigurer dynamiquement le composant de localisation en fonction du niveau d'énergie et de la confiance dans les systèmes de localisation. Pour aller plus loin, nous souhaiterions pouvoir adapter ce composant en fonction d'autres aspects non-fonctionnels, en particulier en fonction des événements dans son environnement. Nous étendons l'architecture du composant de localisation et considérons que le véhicule se déplace sur des routes dédiées au CyCab. Sur ces routes, il y a des zones identifiées où le GPS ne fonctionne pas correctement (tunnels par exemple). Dans ces zones (appelées zones Wifi), l'utilisation du GPS n'est plus nécessaire. L'entrée et la sortie de ces zones sont détectées grâce à des capteurs placés sur les routes dédiées. Dans notre architecture, un composant *Sensor_Area* est ajouté à notre composant de localisation. Nous souhaiterions exprimer que lorsque le composant de localisation se trouve dans une zone Wifi, le composant *GPS* sera supprimé plutôt que le composant *Wifi*. Néanmoins, le composant *GPS* n'est supprimé que si le véhicule est entré dans une zone Wifi et qu'il n'en est pas encore sorti. Malheureusement, cet aspect non-fonctionnel ne peut pas être exprimé dans le cadre des politiques d'adaptation introduites dans (Chauvel *et al.*, 2009). Notre proposition est d'étendre ce cadre en y incluant les logiques évènementielles proposées dans (Lee *et al.*, 1999; Tournier *et al.*, 2005; Gonnord *et al.*, 2009).

5.1. *La logique EDL (Event Definition Language)*

Dans (Lee *et al.*, 1999), pour spécifier les comportements d'un système en fonction d'un ensemble de besoins, les auteurs ont défini deux logiques qui portent sur la notion d'évènement. Une de ces logiques, appelée MEDL, permet d'exprimer les besoins qui doivent être vérifiés lors de l'exécution du système. L'autre logique, appelée PEDL, permet de détecter les évènements du système. Ces deux logiques sont issues de EDL qui est une logique temporelle permettant de prendre en compte des dépendances entre des conditions et des évènements.

Dans (Tournier *et al.*, 2005; Gonnord *et al.*, 2009), les auteurs proposent une architecture logicielle à composants pour la gestion de la qualité de service au niveau des ressources, appelée Qinna. Cette architecture intègre plus particulièrement une extension de la logique EDL, qMEDL. Cette logique prend en compte à la fois un aspect temporel (grâce aux événements E et aux conditions C) et un aspect quantitatif (grâce à Q).

Syntaxe de qMEDL (Gonnord *et al.*, 2009) La syntaxe de qMEDL est définie par :

$C ::= [E, E) \mid C \,\&\&\, C \mid C \mid\mid C \mid Q \bowtie K \mid c$
$E ::= e \mid start(C) \mid end(C) \mid E\ when\ C \mid E \,\&\&\, E \mid E \mid\mid E$
$Q ::= v \mid Q \,\square\, Q$

où $\bowtie \in \{\leqslant, =, <, \ldots\}$, c est une condition dans $\mathcal{C} = \{c, c_1, c_2, \ldots\}$, v est une variable, K est une constante, e est un évènement dans $\mathcal{E} = \{e, e_1, e_2, \ldots\}$, $\square \in \{+, -, *, /\}$.

Sémantique (Gonnord *et al.*, 2009) La sémantique est définie sur les modèles de la forme $M = (\Sigma, \tau, L_C, L_E)$ où :

– $\Sigma = \{\sigma_1, \sigma_2, \ldots\}$ un ensemble d'états d'observation du système ;

– $\tau : \Sigma \to \mathbb{N}$ associe l'instant d'observation à chaque état ;

– $L_C : \Sigma \times \mathcal{C} \to \mathbb{B}$ est l'évaluation des conditions à chaque état ;

– $L_E : \Sigma \times \mathcal{E} \to \perp \uplus D$ où $D = \prod_{e_i \in \mathcal{E}} D_{e_i}$ donne l'information à propos de l'évènement e à l'état σ_k. Si $L_E(\sigma_k, e) = \perp$, il n'y a pas d'évènement e. A l'inverse, $L_E(\sigma_k, e)$ donne les « valeurs » associées avec l'évènement e, prises dans le domaine D_e.

Etant donnés le modèle M et un instant t, la satisfaction de C (respectivement de E), noté $M, t \models C$ (resp. $M, t \models E$), est récursivement définie par :
Cas de base

– $M, t \models c_k$ si et seulement si la condition c_k est vraie à l'instant $t - 1$;

– $M, t \models e_j$ si et seulement si l'évènement e_j est déclenché à l'instant t.

Récurrence

– $\&\&$, $||$ sont respectivement le et/ou de la sémantique de la logique classique.

– $start(C)$ est un évènement qui est déclenché lorsque C devient vraie.

– $end(C)$ est un évènement qui est déclenché lorsque C devient fausse.

– $E\ when\ C$ est déclenché lorsque E est déclenché est que C est vraie.

– La condition $[E_1, E_2)$ est vraie à l'instant t s'il existe un instant t_0 précédent t pour lequel E_1 a été déclenché et pour tout t' tel que $t_0 \leqslant t' \leqslant t$, E_2 n'a pas été déclenché.

Par rapport à la logique floue, qMEDL ne permet pas d'exprimer des aspects qualitatifs. Par exemple, il nous est impossible d'écrire `when power is 'high'`. Par contre, comme qMEDL prend en compte les aspects quantitatifs, nous pourrons exprimer la propriété précédente par `when power >= 67` grâce à la règle $Q \bowtie K$. La logique qMEDL permet d'exprimer un aspect non-fonctionnel portant sur les dépendances entre conditions et évènements. Nous pourrons alors exprimer la propriété sur la détection de zones Wifi. Ainsi pour exprimer que « l'énergie est faible et que le composant de localisation est dans une zone Wifi », nous pourrons utiliser la propriété `[entry,exit) && power < 33`. Les appels au composant *Sensor_Area* sont spécifiés grâce aux événements `entry` et `exit`. Une des règles architecturales de la politique d'adaptation concernant le composant *Wifi* s'écrirait alors comme suit :

```
when [entry,exit) || (power >= 33)
    if size($context/child::wifi) > 0
    then utility of removeWifi is 'low'
```

Cela permet d'exprimer que le composant *Wifi* ne doit pas être supprimé lorsque l'énergie est suffisante et supérieure ou égale à 33 % ou lorsque l'énergie est faible mais le composant de localisation est dans une zone Wifi. Le lecteur peut constater que les deux logiques doivent être utilisées conjointement.

5.2. *Proposition de politique d'adaptation prenant en charge la logique qMEDL*

Pour pouvoir utiliser qMEDL, il faut l'intégrer à l'extension Fractal dédiée aux politiques d'adaptation. Nous souhaitons utiliser cette logique pour détecter le besoin de reconfiguration du composant de localisation. En ce qui concerne l'utilité de reconfigurer l'architecture, nous utiliserons toujours la logique floue. Voici une politique d'adaptation possible concernant l'ajout ou la suppression du composant *GPS* et prenant en compte qMEDL :

```
policy WithGPS
is
  reconfiguration addGPS is "addGPS.fscript"
  reconfiguration removeGPS is "removeGPS.fscript"

  property power : Real
  sensor is getPower on Ctrl

  property entry : EVENT
  property exit : EVENT

  when power >= 33
    if size($context/child::GPS) == 0
    then utility of addGPS is 'high'

  when ([entry,exit) && power < 33)
    if size($context/child::GPS) == 0
    then utility of addGPS is 'low'

  when ([entry,exit) && power < 33)
    if size($context/child::GPS) > 0
    then utility of removeGPS is 'high'

  when power >= 33
    if size($context/child::GPS) > 0
    then utility of removeGPS is 'low'

end policy
```

Nous envisageons d'implanter ce type de politiques d'adaptation et donnons ici quelques pistes pour intégrer la logique qMEDL. Dans ce but, plusieurs étapes seront nécéssaires :

– Nous devrons ajouter un mécanisme nous permettant de capturer les évènements et les valeurs des différentes variables au cours de l'exécution. Pour cela, nous utiliserons un intercepteur Fractal (Bruneton *et al.*, 2006) qui enregistrera les données dans un fichier.

– Le langage de spécification des politiques d'adaptation devra être étendu pour pouvoir reconnaître le langage de la logique qMEDL.

– Le lien entre les deux étapes précédentes devra être établi, en utilisant le fichier généré par l'intercepteur pour vérifier si les propriétés qMEDL sont satisfaites.

6. Conclusion

Cet article décrit les politiques d'adaptation appliquées à un composant de localisation. Nous utilisons l'extension Tangram4Fractal pour décrire ces politiques basées sur la logique floue. Nous avons proposé deux politiques d'adaptation pour ce composant et nous avons simulé son fonctionnement. Les expérimentations mises en place montrent le comportement du composant de localisation en fonction de différents aspects non-fonctionnels tels que l'énergie et la confiance. Nous proposons ensuite d'étendre ces politiques à un autre aspect permettant la prise en compte d'évènements. L'implantation de cette extension est en cours. Actuellement, nous explorons également la faisabilité de notre proposition dans le cadre de l'implémentation Think de Fractal.

Au sein de projet ANR TACOS, d'autres travaux de spécification du composant de localisation sont en cours. Parmi ces travaux, nous souhaitons vérifier l'adéquation de notre approche par rapport aux besoins spécifiés à l'aide de KAOS (Knowledge Acquisition in autOmated Specification). Les autres spécifications proposées telles que CSP||B (Colin *et al.*, 2008a; Colin *et al.*, 2008b), Event B (Mashkoor *et al.*, 2009), ne prennent pas en compte l'aspect dynamique des reconfigurations mais proposent la vérification de propriétés sur des architectures statiques.

Remerciements

Nous remercions les rapporteurs pour leurs remarques constructives.

7. Bibliographie

Bruneton E., Coupaye T., Leclercq M., Quéma V., Stefani J.-B., « An Open Component Model and its Support in Java », *Seventh International Symposium on Component-Based Software Engineering (CBSE-7)*, n° 3054 in *LNCS*, p. 7-22, May, 2004.

Bruneton E., Coupaye T., Leclercq M., Quéma V., Stefani J.-B., « The Fractal Component Model and its Support in Java », *Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, n° 36(11-12) in *LNCS*, 2006.

Chauvel F., Méthodes et outils pour la conception de systèmes logiciels auto-adaptatifs, PhD thesis, Université de Bretagne Sud, September, 2008.

Chauvel F., Barais O., Plouzeau N., Borne I., Jézéquel J.-M., « Composition et expression qualitative de politiques d'adaptation pour les composants Fractal », *Actes des Journées nationales du GDR GPL 2009*, Toulouse, France, janvier, 2009.

Colin S., Lanoix A., Kouchnarenko O., Souquières J., « Towards Validating a Platoon of Cristal Vehicles using CSP||B », *12th International Conference on Algebraic Methodology and Software Technology (AMAST 2008)*, n° 5140 in *LNCS*, Springer-Verlag, p. 139-144, July, 2008a.

Colin S., Lanoix A., Kouchnarenko O., Souquières J., « Using CSP||B Components : Application to a Platoon of Vehicles », *13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2008)*, LNCS, Springer-Verlag, September, 2008b.

Conan D., Rouvoy R., Seinturier L., « Scalable Processing of Context Information with COSMOS », *DAIS*, p. 210-224, 2007.

David P.-C., Ledoux T., « WildCAT : a generic framework for context-aware applications », *MPAC*, p. 1-7, 2005.

Gonnord L., Babau J.-P., « Quantity of Resource Properties Expression and Runtime Assurance for Embedded Systems », *7th ACS/IEEE International Conference on Computer Systems and Applications, AICCSA*, May, 2009.

Lee I., Kannan S., Kim M., Sokolsky O., Viswanathan M., « Runtime Assurance Based On Formal Specifications », *PDPTA*, p. 279-287, 1999.

Mashkoor A., Jacquot J.-P., Souquières J., « B événementiel pour la modélisation du domaine : application au transport », *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'2009)*, Toulouse France, p. 17-35, 2009.

Navas J.-F., Babau J.-P., Lobry O., « Minimal yet effective reconfiguration infrastructures in component-based embedded systems. », *SINTER'09*, ACM, p. 41-48, 2009.

Pedrycz W., « Fuzzy control and fuzzy systems. Second extended edition, Wiley, New York, USA », *Neurocomputing*, vol. 10, n° 1, p. 97 - 98, 1993.

Polakovic J., Mazare S., Stefani J.-B., David P.-C., « Experience with Safe Dynamic Reconfigurations in Component-Based Embedded Systems », *CBSE*, p. 242-257, 2007.

Szyperski C., *Component Software Beyond Object-Oriented Programming*, Addison-Wesley and ACM Press, 1999.

Tournier J.-C., Babau J.-P., Olive V., « Qinna, a Component-Based QoS Architecture. », *in* G. T. Heineman, I. Crnkovic, H. W. Schmidt, J. A. Stafford, C. A. Szyperski, K. C. Wallnau (eds), *CBSE*, vol. 3489 of *Lecture Notes in Computer Science*, Springer, p. 107-122, 2005.

Zadeh L., « The concept of a linguistic variable and its application to approximate reasoning », *Information Sciences*, vol. 1, p. 119-249, 1975.